SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

PISTEPILVIDATAN VISUALISOINTI VIRTUAALITODELLISUUSYMPÄRISTÖSSÄ

OSANA TEOLLISUUDEN SUUNNITTELUSOVELLUSTA

TEKIJÄ: Pentti Pärnänen

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Tietotekniikan koulutusohjelma	
Työn tekijä Pentti Pärnänen	
Työn nimi Pistepilvidatan visualisointi virtuaalitodellisuusympäristössä osana teollisuuden suunnittelusovellusta	
Päiväys 16.5.2018	Sivumäärä/Liitteet 30/0
Ohjaajat TKI-asiantuntija Mikko Pääkkönen, TKI-asiantuntija Mikko Laasanen	
Toimeksiantaja/Yhteistyökumppani 3D Talo Finland Oy	

Tiivistelmä

Opinnäytetyön lähtökohtana oli tutkia mahdollisuuksia suoran pistepilvidatan visualisoinnin lisäämiseksi toimeksiantajayrityksessä kehitteillä olevaan virtuaalitodellisuuspohjaiseen teollisuuden suunnittelusovellukseen. Sovelluksen kehitystyön peruspiirteet olivat tutkimuksen alkuhetkellä jo lyöty lukkoon, ja tietyt rajoitteet pistepilvidatan käytön suhteen olivat näin tiedossa.

Kyseinen sovellus käyttää Unity-pelimoottoria, perustuu 3D-mallien tuontiin ja vapaaseen sijoittamiseen käyttäjän tekemänä ja tukee useaa samanaikaista käyttäjää. Ensisijaisena käyttöliittymänä on virtuaalilasit käsiohjaimineen (kehitystyössä ensisijaisena HTC Vive). Pääasiallisena ohjelmointikielenä toimintalogiikan osalta on C# Unityn rajapintojen ja pelimoottorin tukeman .NET -version mukaisine lisäyksineen, ja lisäksi työssä käytetään Unityn plugin-rajapinnan kautta C++ -kielellä kirjoitettua rajapintaa ulkoisiin kirjastoihin sekä HLSL-kieltä (high level shading language) näytönohjaimen suoritinyksiköiden hyödyntämisessä.

Tehtävänantona oli tutkia yleisesti mahdollisuuksia, rajoitteita ja parhaita toteutusmenetelmiä pistepilvidatan visualisointiin pohjaselvityksenä toimeksiantajayrityksen tuotekehitykseen. Jos näyttäisi, että toivotut ominaisuudet ovat toteutettavissa siten että ne voidaan yhdistää julkaisukelpoiseen tuotteeseen jatkuisi näiden ominaisuuksien kehitys työsuhteena. Ensisijaisesti kyseessä olisi siis alustava selvitys, tutkimusprojekti, jonka tulokset ratkaisisivat jatkon. Koska selvisi, että käytännön toteuttamiseen kannattaa lähteä, jatkui projekti julkaisukelpoisen tuotteen kehityksenä varsin pian pohjaselvityksen alkamisen jälkeen. Opinnäytetyön palautushetkellä tuotekehitys aiheen parissa jatkuu kohti julkaisukelpoista tuotetta, ja tämän jälkeen normaalin tuotteen päivitys- ja ylläpitokehityksen muodossa.

Avainsanat
Pistepilvet, Virtuaalitodellisuus, Unity, PCL, C#, C++, HLSL, Shader

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author Pentti Pärnänen			
Title of Thesis Visualization of point cloud data in virtual reality as a part of industrial design software			
Date	16 May 2018	Pages/Appendices	30/0
Supervisors Mr Mikko Pääkkönen, RDI Specialist and Mr Mikko Laasanen, RDI Specialist			
Client Organisation /Partners 3D Talo Finland Oy			

Abstract

The purpose of this thesis was to research the possibilities for adding direct visualization of point cloud data as a part of virtual reality industrial design software. As a part of the software being already put in development, it should follow some basic principles of the larger project.

The software project is based on Unity engine and it allows the user to import 3D models to place them freely in a virtual design space. It also contains support for multi-user sessions over the Internet. As the primary user interface, it has support for Virtual Reality hardware (HTC Vive as the main development hardware). The game engine level functionality is programmed with C#, and a connection to C++ parts is made with a dll plugin for Unity. The result for visualization with VR equipment is rendered with the HLSL (high level shading language) shader code, making use of the shading units of graphics hardware.

The means, limits and best architectural choices for visualizing massive point clouds were researched while the project progressed. As the early stage research was successful, the project continued as normal product development. While at early stage, the primary goal was testing if point cloud direct rendering would be technologically possible at all considering hardware limitations, at the later stage user experience and optimizations were more important. Also combining the user tools of the main application with the functionality developed during the research started taking an important role.

As a result of this thesis, a working point cloud direct rendering system, which uses Unity engine and is usable with virtual reality gear like HTC Vive, is now accomplished. New functionality, optimizations and implementation to the main application will be a later part of the whole research and product development project. As the project is quite massive, it was clear from the beginning that if the research during the thesis work seemed successful, the product development part would continue for a much longer time.

Keywords
Point Clouds, Virtual Reality, Unity, PCL, C#, C++, HLSL, Shader

SISÄLTÖ

1 JOHDANTO

Opinnäytetyön lähtökohtana oli tutkia mahdollisuuksia suoran pistepilvidatan visualisoinnin lisäämiseksi toimeksiantajayrityksessä kehitteillä olevaan virtuaalitodellisuuspohjaiseen suunnittelusovellukseen. Sovelluksen kehitystyön peruspiirteet olivat tutkimuksen alkuhetkellä jo lyöty lukkoon, ja tietyt rajoitteet pistepilvidatan käytön suhteen olivat näin tiedossa.

Sovellus on tarkoitettu käytettäväksi mm. rakennusteollisuudessa ja ympäristönsuunnittelussa arkisena työvälineenä, teollisuuden kokoonpanolinjojen ja muiden teollisuus- sekä työskentelytilojen suunnittelua helpottamaan ja havainnollistamaan. 3D-mallinnettujen objektien ja olemassa olevien fyysisten ympäristöjen visualisointi virtuaalitodellisuusympäristössä on hyödyllistä näiden lisäksi monissa muissakin yhteyksissä. Vastaavaa sovellusta ei ole vielä markkinoilla erityisen toimivana, ja tämän kaltaiselle työkalulle on ollut tarvetta yrityksen asiakkaiden keskuudessa.

Virtuaalitodellisuusympäristössä käytettävien erilaisten suunnittelu- ja visualisointisovellusten käyttö on teollisuudessa nopeasti kasvava trendi. Näiden avulla voidaan saavuttaa hyvinkin suuria suoria kustannussäästöjä, ja potentiaalisesti välttyä valtavilta välillisiltä kustannuksilta havaitsemalla mahdolliset ongelmat asennustöissä, tuotantolinjojen suunnittelussa, työskentelytilojen ergonomiassa yms. jo ennakkoon. Objekteja, joita ei ole vielä olemassa fyysisinä malleina, voidaan sovittaa ennakkoon osaksi laajempia kokonaisuuksia ja olemassa oleviin rakennettuihin tai luonnonympäristöihin huomattavasti havainnollisemmin kuin tietokoneen näytöllä tai paperilla niitä tarkastellen. Lisäksi teknisten dokumenttien lukemiseen ja perinteisten suunnitteluohjelmistojen käyttöön perehtymättömät loppukäyttäjät voivat vertailla ennakkoon esimerkiksi työskentelytilojen suunnitelmista useampia vaihtoehtoja keskenään, antaa niistä kommentteja ja kokeilla omatoimisesti pienempiä muutoksia.

Kyseinen sovellus käyttää Unity-pelimoottoria, perustuu 3D-mallien tuontiin ja vapaaseen sijoittamiseen käyttäjän tekemänä ja tukee monen käyttäjän yhtäaikaista käyttöä. Ensisijaisena käyttöliittymänä on virtuaalilasit käsiohjaimineen, kehitystyössä käytettiin pääasiassa HTC Vive -laitteistoa. Koska sovelluksen runkona toimii Unity-pelimoottori, on ohjelmointikielenä toimintalogiikan osalta C# Unityn rajapintojen ja pelimoottorin tukeman .NET -version mukaisine lisäyksineen. Lisäksi työssä käytetään Unityn plugin-rajapinnan kautta kytkettyä C++ -kielellä kirjoitettua dll-liitännäistä. Sen avulla saadaan pääsy ulkoisiin C++ -kirjastoihin pistepilven prosessoimiseksi, ja dll:n toiminnallisuutta voidaan kutsua sen ulkopuolelle paljastetuilla funktioilla tarvittaessa Unityn lisäksi vaikkapa yksinkertaisemmasta työpöytäsovelluksesta tai ASP.NET:lla toteutetuista web-sovelluksista mikäli se on tarpeen esimerkiksi pistepilvidatan tuonnin sekä esiprosessoinnin yhteydessä.

HLSL-kieltä (high level shading language) käytetään pistepilven piirrossa näytönohjaimen grafiikkasuorittimella, ja tiettyjen suoraan pistepilvidatan ominaisuuksiin liittyvien laskutoimitusten toteuttamiseen. Tässä toteutuksessa edellytetään vähintään DirectX:n version 11 tukea käytettävältä näytönohjaimelta.

Tehtävänantona oli tutkia yleisesti mahdollisuuksia, rajoitteita ja parhaita toteutusmenetelmiä pistepilvidatan visualisointiin pohjaselvityksenä toimeksiantajayrityksen tuotekehitykseen. Jos näyttäisi, että toivotut ominaisuudet ovat toteutettavissa siten että ne voidaan yhdistää julkaisukelpoiseen tuotteeseen jatkuisi näiden ominaisuuksien kehitys työsuhteena. Ensisijaisesti kyseessä olisi siis alustava selvitys, tutkimusprojekti, jonka tulokset ratkaisisivat jatkon. Koska selvisi, että käytännön toteuttamiseen kannattaa lähteä, jatkui projekti julkaisukelpoisen tuotteen kehityksenä varsin pian pohjaselvityksen alkamisen jälkeen.

1.1 Tilaaja

Työn tilaajana toimi kuopiolainen ohjelmistoyritys 3D Talo Finland Oy, joka on keskittynyt erityisesti virtuaali- ja lisätyn todellisuuden hyötysovelluksiin. Kehitykseen osallistui myös mekaniikkasuunnitteluun, fotogrammetriaan ja laserkeilaukseen erikoistunut Solidcomp Oy, joka toimii samoissa tiloissa 3D Talon kanssa. Monet keskeisistä käsitteistä ja kysymyksenasetteluista joita tässä opinnäytetyössä kohdattiin perustuvat heidän erikoistumisalueidensa konsepteihin.

Työntekijöitä 3D Talo Finland Oy:ssa, Solidcomp Oy:ssa ja kolmannessa tiiviissä yhteistyössä samoissa tiloissa toimivassa Caffeine Overdose Oy:ssa on yhteensä noin 30. Tässä kyseisessä projektissa työskentelee ihmisiä kaikkien näiden kolmen yrityksen (”osaston”) alta, yhteensä vajaa puolet yhteenlasketusta henkilöstöstä. Osa heistä on tässä projektissa mukana tiiviimmin, osa taas tekee tai on tehnyt pieniä osia keskittyen selvästi suurimman osan työajastaan muihin hankkeisiin.

1.2 Opinnäytetyön rajaus

Opinnäytetyöhön melko selkeästi rajattuna osa-alueena kokonaisprojektista oli pistepilvidatan esittämisen suunnittelu ja toteutus. Muut sovelluksen kehityksessä työskentelevät eivät siis suoraan puutu aiheisiin mitkä liittyvät tekniseen ja käytettävyyspuoleen tältä osa-alueelta. Projektin alkuvaiheessa ennalta määriteltyjen tavoitteiden suhteen etenemisestä raportoitiin säännöllisesti kokonaisuudesta vastaaville ja tiedusteltiin uusia prioriteetteja. Kehityksen aikana aiempaa karkeaa edistymissuunnitelmaa muuttamaan pakottavia ongelmakohtia kohdattaessa järjestettiin pienimuotoisia palavereja, joihin osallistuivat ne eri projektin osa-alueista vastaavat kehittäjät ja suunnittelijat, joiden näkemys oli sillä kertaa relevanttia.

Opinnäytetyön rajauksen sisään kuului kaksi toisistaan eriävää lähtökohtaa: Toisaalta tekniset ongelmat, rajoitteiden ratkaisu tai kiertäminen, ja toisaalta käytettävyys. Teknisten ongelmakohtien suhteen on edetty tutkimalla asiaa käsitteleviä tutkimuksia ja koodiesimerkkejä tai teknisiä demoja, sekä jonkin verran myös saman kaltaisia ominaisuuksia sisältäviä valmiita tuotteita. Käytettävyyspuolella tärkeimpänä prioriteettina on kuinka pistepilvidataa olemassa olevat tekniset rajoitteet huomioiden olisi käyttäjäystävällistä esittää virtuaalitodellisuuslaitteistojen kanssa, ja mitä säätöjä sen suhteen on järkevää tuoda käyttäjän saataville sekä suoraan vr-käyttöliittymässä että ennen käytön aloittamista.

1.3 Lyhenteet ja määritelmät

VR, Virtuaalitodellisuus = Menetelmä missä luodaan käyttäjälle illuusio keinotekoisesti luodun ympäristön sisällä olemisesta syöttämällä käyttäjän kummallekin silmälle stereonäön teoriaa hyödyntävällä tavalla hieman erilaiset kuvat

LIDAR = ”Light Radar”, Light Detection And Ranging (menetelmä), Laser-keilaus (menetelmä), Laser-skannaus (menetelmä)

RGB = ”Red-Green-Blue”, menetelmä ilmaista ns. RGB-värijärjestelmän perusväreille lukuarvoja antamalla mikä tahansa väri minkä tietokonelaitteisto pystyy esittämään

Polygonimalli = Kolmiulotteisessa tietokonegrafiikassa kolmiulotteinen objektin malli, kappale mikä koostuu kaksiulotteisista geometrisistä primitiiveistä eli polygoneista

Verteksi = Yksittäinen piste tietokonegrafiikassa. Yleisesti verteksejä käytetään polygonien muodostamisessa niiden kulmapisteinä, mutta piirrettäessä pistepilvidataa sellaisenaan yksittäinen piste on myös verteksi. Muodostettaessa pistepilven pisteistä geometrisia primitiivejä luodaan yhden verteksi sijainnin perusteella useampi verteksi halutuin keskinäisin sijainnin riippuvuuksin primitiivi-polygonin muodostamiseksi

Unity = Opinnäytetyön aiheen sisältävässä ohjelmistoprojektissa käytettävä pelimoottori

GameObject = Unity-pelimoottorin yleisen tason perusobjekti, joka ei sisällä itsessään mitään toiminnallisuutta. Nämä toimivat kehyksinä komponenteille (Component) joilla varsinainen toiminnallisuus toteutetaan

Pelimoottori = Valmis ohjelmistokehys kehitystyökaluineen jolla voidaan toteuttaa pelejä yms. pelillisiä ominaisuuksia sisältäviä ohjelmistoja ilman että kehittäjän tarvitsee keskittyä matalan tason perusrutiinien ohjelmointiin

Overhead = Ylimääräinen resurssien tarve / kasvanut suoritusaika jonkin muun ohjelmistokomponentin kautta, kun ollaan toteuttamassa tiettyä toimintoa tai operaatiota

LOD = ”Level Of Detail”, menetelmä millä vähennetään piirrettävän aineiston kompleksisuutta käytettäväksi tilanteissa, joissa täysi tarkkuus ei ole tarpeen. Esimerkiksi peleissä kauas katsojasta jäävien tai muuten vähemmän merkitsevien alueiden matalampi piirtotarkkuus

2 PISTEPILVET

Pistepilvet yleisesti ovat mitä tahansa asiaa kuvaavaa ja missä tahansa muodossa tallennettua dataa, joka koostuu suuresta joukosta yksittäisiä pisteitä ja joiden esitysmuodoksi on tarkoitettu kolmiulotteinen koordinaatisto (Levoy ja Whitted 1985, 3-4). Tässä yhteydessä pistepilvet kuvaavat fyysistä ympäristöä. Jokainen yksittäinen piste on tässä projektissa käsiteltävissä aineistoissa aina yksittäinen näyte ilman sekä kiinteän materiaalin tai nesteen rajapinnassa.

Sovelluksen esittämäksi tarkoitettu pistepilvidata saadaan skannaamalla fyysistä ympäristöä laserkeilaimella (LIDAR). Kyseessä on yleisimmin jalustalle asetettava laite, joka skannaa ympäristönsä mittaamalla piste kerrallaan etäisyyttä lasersäteen heijastuksen avulla, ja vaihtamalla jokaisen mittauksen jälkeen aina mittaussäteen suuntakulmaa hieman. Saadut etäisyydet lasketaan mittalaitteen paikalliseen koordinaatistoon jolloin ne saavat XYZ-arvot metreinä. Lisäksi useat laitteet tallentavat pisteiden heijastuksen voimakkuuden, mikä kuvaa kohdatun materiaalin heijastavuutta mittaavan laserin aallonpituudella. Tyypillisesti tällainen laite myös päättelee pisteille väriarvot näkymästä otettujen valokuvien perusteella, ja näin saadut väriarvot tallennetaan pisteille RGB-värikomponentteina.

Tämän tuloksena seurattaessa yleisesti käytössä olevaa työnkulkua, on LIDAR-laitteiston käyttäjällä tallennettuna useista mittausasemista saadut erilliset teksti- ja numerodataa sisältävät tiedostot, joiden suuruusluokka on kymmeniä miljoonia yksittäisiä pisteitä per tiedosto. Näistä löytyy jokaiselle pisteelle XYZ-sijainti metreinä, takaisinheijastuksen voimakkuus, ja RGB-väriarvot. Mittausoperaation päätteeksi mittaaja rekisteröi nämä erilliset tiedostot keskenään yhteen ja samaan koordinaatistoon, mikä tapahtuu kirjoittamalla jokaisen tiedoston header-osioon muunnosmatriisi mitä soveltamalla kyseisen tiedoston paikallisen koordinaatiston mukaiset pisteiden sijainnit saadaan muutettua kaikkien mittauspisteiden tuloksille yhteiseen koordinaatistoon.

3 TEKNISET LÄHTÖKOHDAT

Projekti aloitettiin alun perin tutkimushankkeen lähtökohdasta, ja tällöin oli toimeksiantajan ennakoimana eräitä teknisiä rajoituksia. Koska tavoitteena oli alun perinkin tutkia mahdollisuuksia tutkittavan toiminnallisuuden yhdistämiseksi kehitteillä olevaan tuotteeseen, koskivat tuotteen osalta lukkoon lyödyt teknisen toteutuksen yksityiskohdat yhtäläisesti myös tutkimuksen osuutta. Toivottua oli kuitenkin myös tiedon kerääminen sen varalta, ettei toivottu toiminnallisuus olisi ollut toteutettavissa alkuperäisten määritysten mukaisesti, ja poimien myöhemmin mahdollisesti käyttökelpoista tietoa**.**

3.1 Pistepilvien renderointi

Pistepilvien renderointi suoritetaan näytönohjaimella Shader Model 5.0 mukaisessa shader-koodissa, siten että pisteiden data siirretään näytönohjaimen muistissa olevalle muistialueelle ja piirretään näytölle suoraan sieltä ilman että itse muun sovelluksen käyttämä Unity-pelimoottori varsinaisesti enää puuttuu tähän osa-alueeseen datan siirron jälkeen. Yksi aiemmin mainituista, ennakoiduista teknisistä rajoituksista oli ettei pistepilvidatan tuonti piirrettäväksi luomalla siitä ensin meshejä olisi kestävä ratkaisu: Tällä tavalla Unityn sisäisten rajoitteiden kiertämiseksi tarvittaisiin toteutusta monimutkaistavia menetelmiä, ja suorituskyky jäisi muutenkin heikoksi.

Piirtämällä pistepilvet suoraan näytönohjaimen muistiin siirretystä koordinaatti- ja väridatasta saatiin nämä ongelmat kierrettyä minimoitua. Tästä aiheutuu kuitenkin omat ongelmansa: Pistepilvistä piirretty käyttäjälle näkyvä ”kolmiulotteinen valokuva fyysisestä tilasta” ei ole suoraan itse pelimoottorin koodille millään tavalla näkyvää. Siihen ei voi soveltaa mitään Unityn tarjoamia fyysisiä operaatioita, eikä esimerkiksi etäisyyttä pistepilvidatasta piirrettyyn ”seinään” tms. näennäiseen materiaalin rajapintaan voida mitata samalla tavoin kuin pelimoottorin sisäisiin polygon mesheihin voidaan.

3.2 Datan lataus kiintolevyltä

Tiedostoformaattina pistepilvidatan tuonnissa tuetaan tällä hetkellä ptx-tiedostoja, joka on Leican maanpäällisten LIDAR-laitteiden suoraan tukema muoto. Lisäksi muunnos muista eri merkkisten laitteiden kirjoittamista vastaavista tiedostoformaateista onnistuu siihen monilla yleisesti käytössä olevilla työvälineillä. Ptx-tiedostot noudattavat aiemmin kuvattua mallia missä tallennettuna on mittalaitteen paikallisessa koordinaatistossa XYZ-sijainti jokaiselle pisteelle, heijastavuusarvo ja RGB-väriarvo, sekä lisänä header-osiossa muunnosmatriisi, millä nämä sijainnit voidaan muuntaa rekisteröinnin yhteydessä päätettyyn globaaliin koordinaatistoon.

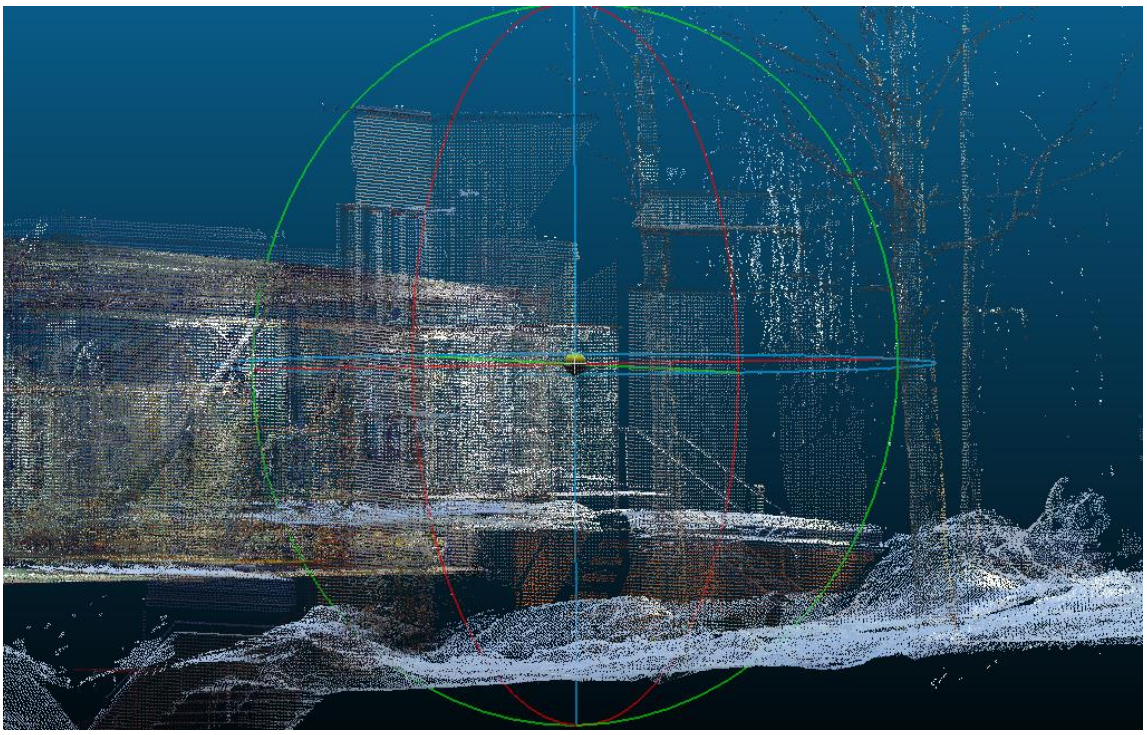
Sovellukselle suunnitelluissa käyttötilanteissa piirrettävä pistepilvidata koostuu tyypillisesti useiden kymmenien tai jopa satojen eri mittauspisteiden datasta. Datan määrä on siis massiivinen, eikä tietokoneen keskusmuisti riitä läheskään sen kaiken pitämiseen tallessa kerralla ja näin ollen dataa on ladattava kiintolevyltä tarpeen mukaan. Tarvitaan siis jokin menetelmä datan siirtämiseksi kiintolevyltä keskusmuistiin ja lopulta näytönohjaimen muistiin tarpeeksi nopeasti, ascii-pohjaisen formaatin ollen lisäksi vaadittuine matriisimuunnoksineen aivan liian hidas.

Mahdollisimman reaaliaikaisen latauksen tarpeeseen ratkaisuna toimii PCL-kirjaston (Point Cloud Library 2018a) kehittämän binääriformaatin käyttäminen ohjelman sisäisenä tallennusformaattina. Tuotaessa ptx-tiedostoja ne muunnetaan PCL-kirjaston formaattiin pcd-tiedostoiksi ohjelman sisäistä käsittelyä varten. Tämä mahdollistaa ptx-tiedostojen ohella suoraan pcd-tiedostojen käytön tuotavana formaattina. Tällä on kuitenkin merkitystä lähinnä kehitysvaiheessa yksittäisten ohjelmakomponenttien nopeamman testauksen mahdollistajana, koska kyseistä formaattia eivät laserkeilaimet ja niiden valmistajat tue suoraan. Näinpä todellisissa käyttötilanteissa ei ole erityisen todennäköistä, että ohjelmiston käyttäjällä olisi erityistä tarvetta tuoda data ohjelmistoon pcd-tiedostoina.

3.3 Pistepilvien piirron käyttäjäkokemus

Perinteisesti tyypillinen tapa hyödyntää pistepilvidataa on muokata sen pohjalta osittain automatisoidusti mutta runsaasti manuaalista työtä vaatien polygonimalleja, joiden esittäminen tietokonesovelluksissa on itsessään suoraviivaista. Nämä mallit ovat tyypillisiä kaikessa 3D-tietokonegrafiikassa ja niiden esittäminen on ollut perusoletuksena niin ohjelmistojen kuin tietokonelaitteistojenkin (näytönohjaimet ajureineen) suunnittelussa.

Haluttaessa visualisoida pistepilvidataa suoraan ilman aikaa vievää manuaalisen työn vaihetta, jossa se muunnetaan polygonimalleiksi, on monissa sovelluksissa tarjolla pistepilvien esittäminen pistejoukkona perinteisellä näytöllä kuten allaolevassa kuvassa (KUVA 1). Tällaisessa käyttötilanteessa menetelmä on nopeaan aineiston tarkistamiseen kohtalaisen toimiva vaikka useissa tilanteissa pistepilvi on läpinäkyvä; Piste on matemaattisen määritelmänsä mukaisesti täysin pistemäinen ilman muita avaruudellisia ulottuvuuksia kuin sijainti, joten matemaattisesti ajateltuna todennäköisyys, että lähempänä oleva piste peittää kauempana olevan on 0 mikäli pisteet todellisuudessa ovat täysin pistemäisiä. Tuo ei tietenkään toteudu tietokoneen näytöllä täysin, koska näytön pikseleillä on aina pinta-ala. Tällainen harva pistepilvi on niiden visualisointiin tarkoitetussa perinteisessä sovelluksessa näytöllä pyöriteltävissä vapaasti, jolloin se kuitenkin hahmottuu paremmin kuvakulman muutosten myötä.



KUVA 1. LIDAR-skannattua pistepilveä CloudCompare-ohjelmiston esittämänä

Virtuaalitodellisuusympäristössä kuvakulman vaihtaminen nopeasti ja ikään kuin ulkopuolelta katsoen ei kuitenkaan ole mahdollista, vaan pistepilven kuvaamaa tilaa katsellaan sisältä päin. Tämän takia työn alla olevassa sovelluksessa tulee tarpeelliseksi muodostaa pistejoukoista reaaliaikaisesti näennäisen yhtenäisiä pintoja.

4 TUTKIMUS TEHTYJEN VALINTOJEN TAUSTALLA

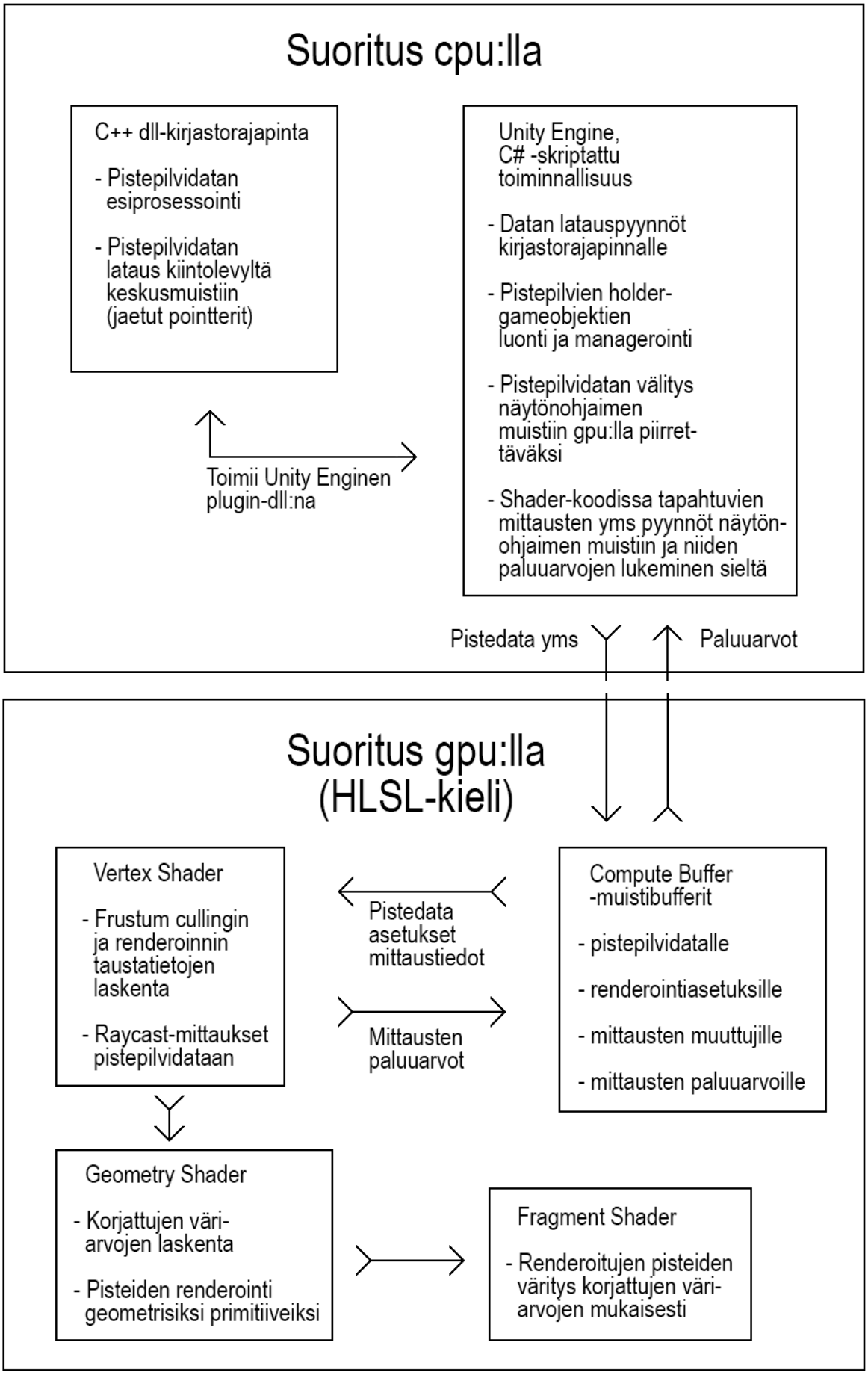
4.1 Laskennan jakautuminen CPU / GPU ja datan kulku

Pistepilvien piirto käyttäjän nähtäväksi vaatii laskentaa kolmessa eri osakokonaisuudessa, joista kaksi ensimmäistä tapahtuu tietokoneen keskussuorittimella (CPU) ja kolmas näytönohjaimen grafiikkasuorittimella (GPU). Alla olevassa kaaviossa (KUVIO 1) on esitetty datan kulku näiden osakokonaisuuksien välillä.

Ensimmäisenä askeleena on C++ -kielellä kirjoitettu ja vahvasti valmiisiin matemaattisiin kirjastoihin pohjautuva, Unity-pelimoottoriin liitetty dll-plugin. Se huolehtii ajonaikaisesta datan lataamisesta kiintolevylle tallennetuista binäärimuotoisista ja valmiiksi prosessoiduista latauslohkokohtaisista datatiedostoista. Lisäksi sen funktioilla toteutetaan myös datan esiprosessointi mikä sisältää lähdetiedostojen pilkkomisen ja yhdistelyn sopiviksi latauslohkoiksi, LOD-tasojen luonnin ja pistetiheyksien tasauksen, metadatan muodostamisen ja kirjoittamisen omaan tiedostoonsa yms. tämän tason toiminnallisuuden.

Varsinkin esiprosessoinnin kohdalla lopullinen ohjelmistotuote tulee tukemaan esiprosessointitoiminnallisuuden käyttöä myös itse Unity-pohjaisen virtuaalitodellisuussovelluksen ulkopuolelta. Tähän tarjoutuu mahdollisuus toteuttamalla esiprosessointi kokonaisuudessaan erilliseen dll:aan keskitetyillä operaatioilla: Näitä funktioita voidaan kutsua helposti myöskin erillisestä työpöytäsovelluksesta (esim. Windows Forms -pohjainen) tai web-sovelluksesta (ASP.NET -pohjainen) ja näin saadaan identtinen lopputulos verrattuna tilanteeseen että niitä kutsuttaisiin itse VR-sovelluksesta käsin. Tällöin erillisessä sovelluksessa luotu, valmiiksi ”importattu” projekti voidaan siirtää suoraan kopioimalla aineisto kansiorakenne säilyttäen käytettäväksi useilla asiakkailla. Esimerkkinä tästä voisi ajatella vaikkapa tilanteen, missä esiprosessointi suoritetaan pilvipalvelussa ja ladataan sieltä useille samaa aineistoa yhteisessä VR-istunnossa käyttäville asiakkaille.

Mahdollisuutta suorittaa datan esikäsittely useasta eri käyttöliittymästä käsin dll:n prosessointitoiminnallisuutta hyödyntäen päätettiin edistää myös pyrkimällä alusta lähtien järjestelmäriippumattomaan ohjelmointitapaan. Käytettävät ulkoiset kirjastot tukevat Windowsin lisäksi myös muita yleisiä käyttöjärjestelmiä (Linux, OS X, BSD-käyttöjärjestelmät), ja käyttöjärjestelmäriippuvaisia rajapintoja ei käytetty. Alkaen C++11-versiosta on kiinnitetty erityistä huomiota käyttöjärjestämäkohtaisten rajapintojen korvaamiseen standardisoiduilla ratkaisuilla, ja C++14 mukaiset lisäykset mahdollistivat C++ -osien kirjoittamisen kokonaisuudessaan järjestelmäriippumattomaksi. Vanhempien C++:n versioiden varassa ollessa tiedostonkäsittelyn ja säikeistyksen toteutus olisi jouduttu kirjoittamaan aina käyttöjärjestelmärajapintojen mukaisesti jokaiselle eri käyttöjärjestelmälle uudelleen, mikä olisi heikentänyt huomattavasti koodin siirrettävyyttä toiselle alustalle. Vaikka tämä osa onkin järjestelmäriippumattomaksi kirjoitettua, keskitytään tässä raportissa yksinomaan Windows-maailmaan, mikä näkyy myös siinä, että selkeyden vuoksi tässä raportissa puhutaan dll-liitännäisestä ja -rajapinnasta.



KUVIO 1. Laskennan jakautuminen keskusprosessorin ja grafiikkasuorittimen välillä, ja datan kulku kiintolevyltä keskusmuistiin sekä sieltä näytönohjaimen muistiin

Unity-pelimoottorin vastuulla on itse virtuaalitodellisuus-käyttöliittymää pyöritettäessä datan pyytäminen tarpeen mukaan dll-liittännäiseltä. Tämä edellyttää käyttäjän sijainnin ja katseen suunnan seurantaa virtuaalimaailman sisällä, erilaisia tarkistuksia sekä kirjanpitoa ladattavaksi tarvittavien, jo ladattujen ja poistettavaksi joutavien latauslohkojen suhteen. Latauslohkoja ladattaessa niille luodaan Unity-pelimoottorin sisällä jokaiselle oma GameObject-olionsa, nämä puolestaan sisältävät instanssin materiaalista mihin gpu:lla eli näytönohjaimen grafiikkasuorittimella suoritettava HLSL-ohjelmakoodi on sidottu.

Erillisiin materiaali-instansseihin sidottuna HLSL-koodissa määritellyt, näytönohjaimen muistissa sijaitsevat muistibufferit ovat toistensa sisällöstä riippumattomat, ja näin Unitysta käsin yksittäisiä GameObjecteja käsittelemällä saadaan ohjattua yksittäisiä pistepilven osia: Kun GameObject kytketään passiiviseksi (gameObject.SetActive(false)) pysyy kyseisen lohkon data näytönohjaimen muistissa mutta sitä ei piirretä eikä se myöskään rasita näytönohjaimen grafiikkasuoritinta, aktivoitaessa se taas tulee jälleen piirrettäväksi, ja poistettaessa kyseinen GameObject vapautuu myös kyseisen datan viemä osa näyttömuistista.

Näytönohjaimen muistista käsin pistepilvidataa lopulta piirrettäessä on shader-ohjelmoinnin yleisten periaatteiden mukaisesti käytettävissä erilliset koodilohkot, joiden tehtävät ovat melko tiukasti määritellyt. Tähän on syynä historiallinen tausta näytönohjaimen shader-yksiköiden kehittymisessä. Toiminnallisuus mitä voidaan missäkin shader-lohkossa suorittaa on ollut hyvin tiiviisti itse grafiikkasuorittimen rautaan sidonnainen. (Fatahalian 2011, 16-20.)

4.2 Pilkkominen ja latausetäisyys

Alusta lähtien oli tiedossa, että sovelluksella olisi tarpeen pyörittää niin massiivisia pistepilvikokonaisuuksia ettei niiden lataaminen kerralla käyttöön olisi mahdollista. Jo keskusmuistin riittävyys asettaisi kovan takarajan kerralla kiintolevyltä ladattujen pisteiden maksimimäärälle. Toisena rajana on tyypillisesti keskusmuistia vähäisempi näytönohjaimen muistin määrä; Kaikki renderointivalmiudessa olevat pisteethän täytyy olla näyttömuistin bufferiin siirrettyinä, sen osakokonaisuuden mukana mikä latausyksiköksi päätetäänkään. Kolmantena tulee näytönohjaimen grafiikkasuorittimen asettama raja: Pisteiden piirto itsessään vaatii tietyn tehon, mutta vaikka osa pisteistä jätettäisiin piirtämättä myös piirtämättä jätettäville pisteille tehtävät tarkistukset tuovat oman osansa kuormituksesta.

Massiivinen määrä tarpeettomia pisteitä näyttömuistin bufferissa ei ainoastaan ole täyttämässä näyttömuistia, vaan myöskin kuormittaa jonkin verran grafiikkasuoritinta silloinkin, kun näitä pisteitä ei piirretä. Tämän ongelman ratkaisemiseksi on olemassa useita mahdollisia menettelytapoja, joista jokaisessa on omat hyvät ja huonot puolensa. Seuraavaksi käydään läpi lyhyesti muutamaa tällaista menetelmää.

4.2.1 Datan vaiheittainen lataus VR-käytön aikana

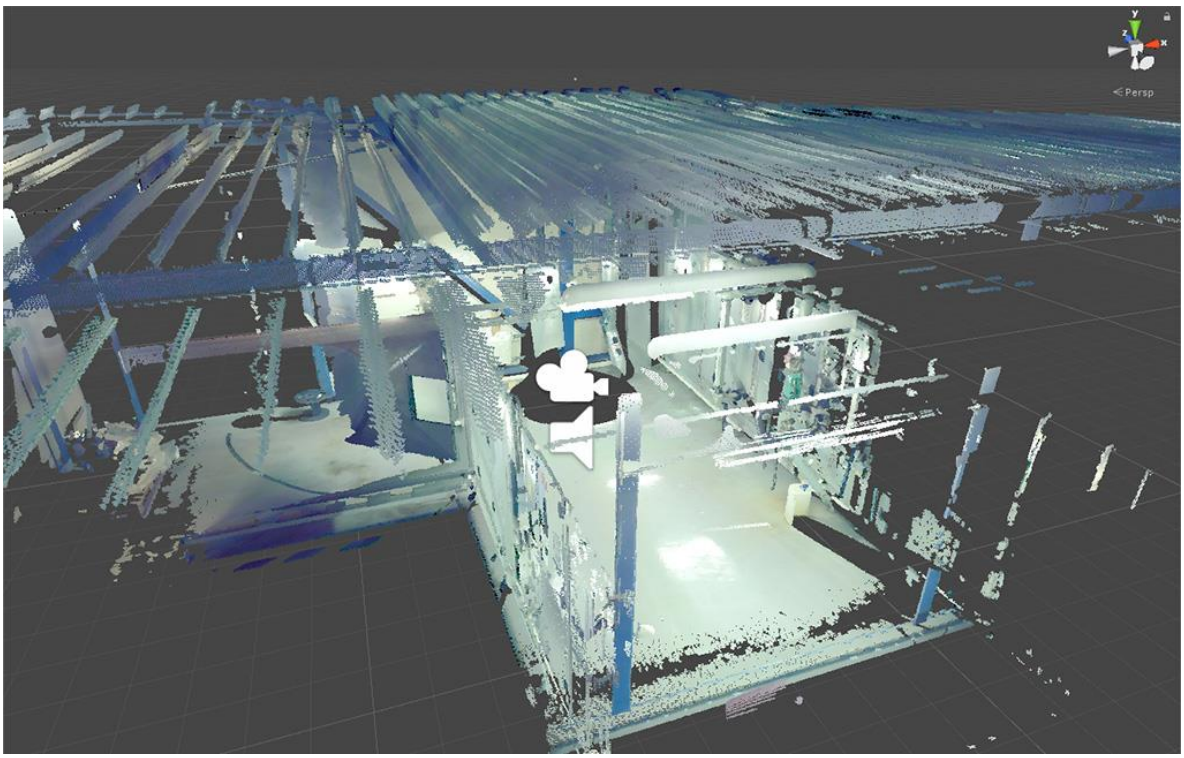
Alkuvaiheessa esitettiin kysymys olisiko mahdollista lukea suuresta pistepilvitiedostosta, mihin on yhdistetty monen LIDAR-mittauspisteen datat, haluttua sijaintia lähelle sijoittuvien pisteiden data lukematta koko tiedostoa kerralla. Osoittautui kuitenkin, ettei tuo ole järkevä lähestymistapa mutta vastaavaa pohjaideaa on kehitelty pidemmälle vietynä.

Tämän kaltainen ratkaisu olisi kirjoittaa pistedata jonkin ns. tilan täyttävän käyrän (esim. Hilbert space-filling curve) mukaan järjestettyyn tietokantaan. Tällöin lähellä toisiaan oleville tietokannan riveille tallennetuilla pisteillä olisi samalla tavalla paikallinen läheisyys toisiinsa myös kolmiulotteisessa avaruudessa. (Wang ja Shan 2005, 3.)

Aiemmasta siirryttiin ajatukseen pilkkoa pistepilven täyttämä koordinaatisto tasakokoisiin kuutioihin jakaen näin alkuperäinen data pieniin osakokonaisuuksiin. Näitä kuution alueelle rajattuja pistepilven osia voitaisiin tämän jälkeen ladata havainnoitsijan sijainnista mitatun etäisyyden mukaan. Ensimmäistä toimivaa testiversiota ja testikäyttöä varten tästä ajatuksesta tehtiin siirtymä mittausasemakohtaiseen lataukseen, mistä seuraavan alaotsikon alla lisää. Lisäksi kuutioidun aineiston käytön toteuttamiseen siirryttiin uudelleen, kun käytettävyystestejä oltiin saatu käytyä läpi.

4.2.2 Mittausasemakohtainen lataus

Materiaalin tuonnin nopeuttamiseksi päädyttiin käyttämään kirjoitushetken kehityshaaran ensimmäisessä toimivassa versiossa yksittäisten mittausasemien tuottamia pistepilvidatan osakokonaisuuksia dynaamisen latausjärjestelmän perusyksiköinä. Data tuodaan sovellukseen tällaisten yksittäisten LIDAR-laitteen sijaintien tuottamien pistepilvitiedostojen joukkona, ja niiden pilkkominen tasakokoisiin kuutioihin sekä jälleenyhdistely olisi useimmissa tilanteissa tarpeettoman aikaavievä operaatio. Lisäksi yksittäinen mittausasema ”näkee” samojen sääntöjen mukaan kuin pistepilvipohjaisessa ympäristössä liikkuva ohjelmiston käyttäjä, joten samassa seinien, lattian ja katon rajaamassa tilassa yksittäisen mittauspisteen pistejoukko on jo valmiiksi käytännönläheisesti järjestynyt. Ohessa on nähtävissä (KUVA 2) yhdestä laserkeilaimen sijainnista taltioitu aineisto; Laite on sijainnut paikassa jossa on nähtävissä kameran symboli.



KUVA 2. Yksittäisen LIDAR-skannauspisteen näkemä alue sisätiloissa, renderoity opinnäytetyön sovelluksen shaderilla ja Unityn editori-ikkunassa tarkasteltuna

Jotta erilaisissa ympäristöissä tapahtuneiden mittausten tuottamien datasettien latausetäisyys käyttäjästä voitaisiin pitää tarkoituksenmukaisen suuruisena, oli tehtyjen havaintojen perusteella järkevää analysoida jokaisen mittausaseman tuottama pistepilvi erikseen ja soveltaa tilastollisia menetelmiä pisteiden sijoittumiskohtien jakauman määrittelyssä. Tämä esimerkiksi mahdollistaa käytännössä vain ahdasta käytävän mutkaa kuvaavan osapistepilven lataamisen, kun käyttäjä on suhteellisen lähellä sen sijaintipaikkaa, ja vastaavasti avarassa ympäristössä sijainneen ja merkittävän osan pisteistä suhteellisen kaukaa mittauspisteestä kattavan osapistepilven lataamisen jo, kun käyttäjä on selvästi kauempana sen keskikohdasta.

4.2.3 Pistepilven kuutiointi ja lataus

Mittausasemakohtainen lataus oli tarkoituksenmukainen nopeasti toteutettavana menetelmänä, kun haluttiin päästä koekäyttämään käyttäjän kokemusta erilaisin pisteiden renderointimenetelmin. Tässä latausmenetelmässä oli kuitenkin suorituskyvyn rajallisuuden ja piirtoetäisyyden suhteen sen verran merkittäviä ongelmia, että seuraavaksi osatavoitteeksi otettiin jo aiemmin suunnitellun pistepilviaineiston kuutioimisen toteuttaminen.

Pistepilvidatan jakaminen tasakokoisiin paloihin on yleisesti käytetyin menetelmä datan vaiheittaiseen lataukseen pistepilviprojekteissa. Esimerkiksi Alberto Nale käyttää toteutuksessaan vastaavan tyyppistä datan paloittelua (tiling) pisteiden avaruudellisen sijainnin perusteella ja listaa tähän liittyviä huomioita (Nale 2014/2015, 26-27) maisteriohjelman työssään.

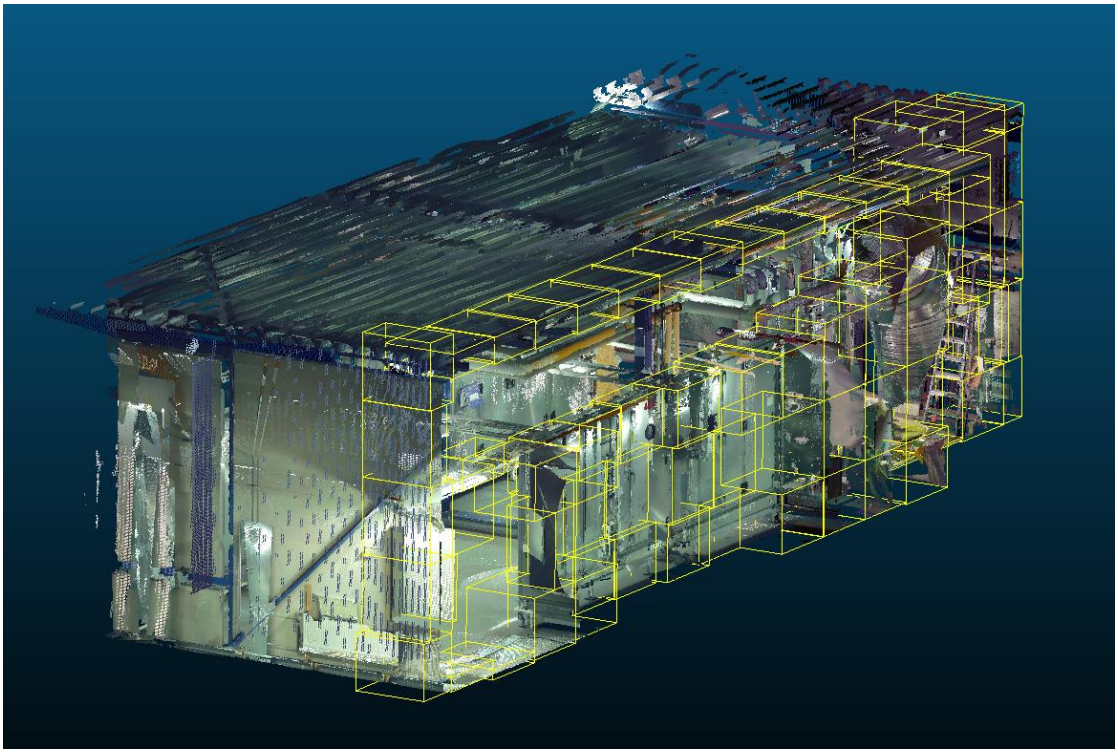
Paloitellessa aineisto tasakokoisiksi kuutioiksi on aineiston lataus tarpeen mukaan sinänsä suoraviivaisempaa, eikä lataustarpeen arvioinnissa tarvita tilastollisia menetelmiä latausyksikön ulottuvuuksien ja pistejakaumien arviointiin. Metadatan kirjoittaminen palojen sijainnista avaruudessa on kuitenkin edelleen välttämätöntä, mutta tuo informaatio on luettavissa datasta kuutioinnin sivussa nopeasti kirjoitettavaksi erilliseen metadatatiedostoon. Vastapainona tälle latausyksiköiden koko kasvaa valtavasti ja lohkojen latausalgoritmit on kirjoitettava kelvollista käyttäjäkokemusta tavoitellessa kokonaan uusiksi. Mittausasemakohtaisten latausyksiköiden kanssa toiminut latausalgoritmi sinänsä toimii itse kuutioinnin onnistumista ja käyttökelpoisuutta testatessa, mutta ei lähellekään asiakkaille toimitettavalta tuotteelta vaadittavalla tavalla.

Lähdeaineiston kuutiointi edellyttää myös eri lähdetiedostojen avaruudellisesti päällekkäin menevien alueiden sisältämien pisteiden yhdistämistä samaan latausyksikkökohtaiseen tiedostoon. Karkeana menetelmänä periaatteen tasolla toimisi se, että kaikki lähdetiedostot ensin yhdistetään yhdeksi pistepilviobjektiksi algoritmin koodissa, ja tämän jälkeen pisteet jaetaan erillisiin pistepilviobjekteihin niiden sijainnin suhteen ja edelleen nämä objektit kirjoitetaan omiksi tiedostoikseen. Tämä ei kuitenkaan toimi käytännössä koska tuon toteutuksen muistintarve suurella lähdeaineistolla olisi liian suuri tietokoneiden keskusmuistin määrään nähden.

Muistin rajallisuus johtaa siis väistämättä väliaikaisten tiedostojen käyttöön kuutiointialgoritmin toteutuksessa. Tällöin on ensiarvoisen tärkeää, että luku- ja kirjoitusoperaatiot kiintolevylle voidaan minimoida ottaen kuitenkin samalla huomioon laskennallisen kuorman ja muistinkäytön pitäminen järkevässä suhteessa niiden viemään aikaan. Myös operaatioiden laskennallisesti raskaan osuuden säikeistäminen on tärkeää koska kyseessä on väistämättä paljon prosessoriaikaa tehokkaimmillakin nykysuorittimilla vaativa operaatio.

Aikaiseksi saatiin loogisesti oikein toimiva ja tehokas, levyoperaatiot minimissä pitävä sekä tehokkaasti säikeistyvä algoritmi. Testattaessa algoritmia 55 eri laserkeilaussijainnin lähdetiedostoilla, joissa on yhteensä noin miljardi pistettä, ja kuutioitaessa se 1m * 1m * 1m kuutioihin joita syntyi yhteensä n. 75000 kappaletta, pysyy suoritinkäyttö modernilla 6-ytimisellä Intelin prosessorilla lähes koko ajan jokaisella ytimellä sadassa prosentissa. Tavallinen SATA-väyläinen ssd-levy ei tule pullonkaulaksi huomattavasta määrästä väliaikaistiedostoja huolimatta. Koska väliaikaistiedostoja kirjoitetaan, luetaan ja poistetaan valtava määrä ja ne ovat kooltaan pieniä, antaa ssd-levy suuren edun nopeista vasteajoista johtuen eikä mekaanista kiintolevyä voi suositella varsinkaan väliaikaistiedostojen kohdeasemana.

Alla olevassa kuvassa (KUVA 3) on osa useiden mittauspisteiden tuottamasta pistepilviaineistosta CloudCompare-ohjelmistossa, joka on ajettu yllämainitun kuutiointialgoritmin läpi. Osa kuutioista on valittuna ohjelmiston tiedostoselain-osassa, jolloin näiden ympärillä on nähtävissä kehykset, loput valitsematta jätetyt taas sulautuvat saumatta toisiinsa. Kehysten sivujen pituuksien suhteen on huomioitava, etteivät ne ole kaikkialla kuutiomaisia, koska kyseinen ohjelmisto piirtää kehykset ainoastaan olemassa olevien pisteiden ympärille. Se ei tiedä kuinka paloitettelualgoritmi toimii vaan käsittää pistepilvitiedostojen ulottuvuudet vain niistä löytyvien pisteiden mukaisesti.



KUVA 3. Säännöllisen kokoiset alueet kattaviin tiedostoihin paloiteltu pistepilviprojektin osa

Tietyn tyyppisissä ympäristöissä kuutioidun aineiston käytöstä on odotettavissa merkittävä suorituskykyetu mittausasemakohtaisiin latausyksiköihin verrattuna. Etu syntyy ympäristöissä, joissa yksittäisen mittauspisteen näkökentän rajojen yhtenevyydestä ihmisen näkökentän kanssa ei ole saatavissa erikseen etua mikä kääntäisi tilanteen päinvastaiseksi. Tällaisia ympäristöjä ovat varsinkin hyvin pitkälle kantavan näkyvyyden olosuhteet kuten ilmasta tapahtunut LIDAR-skannaus ja hyvin suuret avoimet tilat. Yleisellä tasolla kuutioidun materiaalin käytön puolesta puhuu sen sopivuus yleisesti kaikenlaiselle materiaalille, siinä missä mittausasemakohtainen latausyksikkö toimii paremmin vain tarkalleen määritellyissä olosuhteissa.

Maanmittauslaitokselta on saatavissa (Maanmittauslaitos 2018) kattavasti ilmasta skannattua pistepilvimateriaalia ja tälle sekä muille vastaaville materiaaleille on tarpeen luoda myös tuki sovellukseen. Kun sovellus toimii oletusarvoisestikin tasamittaisesti kuutioidulla materiaalilla, ei ilmasta käsin taltioidun materiaalin kanssa tarvita juurikaan muita muutoksia sovellukseen kuin näiden tiedostoformaattien tuki materiaalin tuonnin yhteyteen.

4.3 LOD

Kauempana havainnoijasta olevien pistepilven osien piirtäminen täydellä tarkkuudella ei ole laitteistoresurssien käytön kannalta järkevää. Tämän vuoksi sovellukseen tarvitaan ns. level of detail -systeemi, joka jollain menetelmällä tarjoaa näytönohjaimen piirrettäväksi etäämmälle jäävien latauslohkojen osalta vähemmän yksityiskohtaiset versiot ja vaihtaa tilalle tarkemmat versiot, kun etäisyys pienenee tietyn rajan alle. (Fraiss 2017, 6 ja Nale 2014/2015, 11-12.)

Tämän toteuttamiseen on olemassa useita vaihtoehtoisia menetelmiä, joista jokaisessa on omat etunsa ja haittansa. Osa menetelmistä sisältää useamman LOD-tason luomisen ja tallentamisen etukäteen myöhempää käyttöä varten, osa taas luo niitä reaaliaikaisesti tarpeen mukaan.

Tallentamalla pistepilvidata Hilbertin tilantäyttävän käyrän mukaan järjestettyyn tietokantaan, oltaisiin mahdollisesti saatu suoraan toimivaa karkeamman detaljitason dataa lukemalla vain joka n:s tietokannan rivi. Tämä seuraa suoraan kyseisen käyrän luonteesta, mutta teorian toimivuus jäi kokeilematta, koska tuosta lähestymistavasta datan latauksessa luovuttiin varhaisessa vaiheessa. Tämä olisi vaatinut pistepilvidatalle tietokannan mutta päätettiin pysytellä menetelmässä missä kaikki data pysyy helposti koneelta toiselle kopioitavan kansiohierarkian sisällä.

Yleinen ratkaisumalli pistepilvien LOD-järjestelmien toteuttamisessa ovat etukäteen luodut ja tallennetut eri tarkkuustason versiot datatiedostoista. Erilliset matalamman detaljitason versiot päädyttiin luomaan käyttäen voxelgrid-mallia ratkaisun helpon toteutettavuuden vuoksi kuten myöhemmin kohdassa 5.1.2 (Datan tuonti) kuvataan. Saman tyyppistä ratkaisua käyttää myös Alberto Nale (2014/2015, 31-34). Tämän menetelmän huonona puolena on hieman suurempi tallennustilan tarve kuin reaaliaikaisesti matalamman resoluution versiot täyden tarkkuuden datasta laskien. Käytännössä tämä haitta on merkityksetön, koska mentäessä LOD-tasoissa kohti epätarkempia versioita vähenee näiden tiedostojen koko eksponentiaalisesti.

Toinen reaaliaikaisesti matalamman LOD-tason datan alkuperäisestä laskeva menetelmä, joka kolmiulotteisen koordinaatiston kuutiomaisiin soluihin pilkkomisen osalta muistuttaa jonkin verran voxelgrid-mallia, olisi ollut octree-puurakenteeseen perustuva pisteiden määrän vähentäminen (Fraiss 2017, 6-7). Tämän etuna olisi myös mahdollisuus tuottaa juuri niin tiheä versio ilman etukäteen tehtyä eri LOD-tasojen rinnakkaistiedostojen luontia kuin sillä kertaa satutaan haluamaan, kuten tilantäyttävän käyrän mukaan järjestetystä tietokannasta ”harvalla haulla”. Tällöin kuitenkin menetettäisiin etukäteen luoduilla LOD-tasoilla saavutettava massiivinen ajansäästö kiintolevyltä latauksessa kauempana sijaitsevien osapistepilvien kohdalla, koska joka kerta jouduttaisiin levyltä lataamaan täyden resoluution versio. Hyöty realisoituisi keskus- ja näyttömuistin vähäisempänä täyttymisenä ja gpu:n rasituksen pienemisenä verrattuna täysiresoluutioisen datan käyttöön koko ajan, mutta käytössä olevien massiivisten datamäärien kanssa myös kiintolevyn ajonaikaisen käytön minimointi on tärkeää.

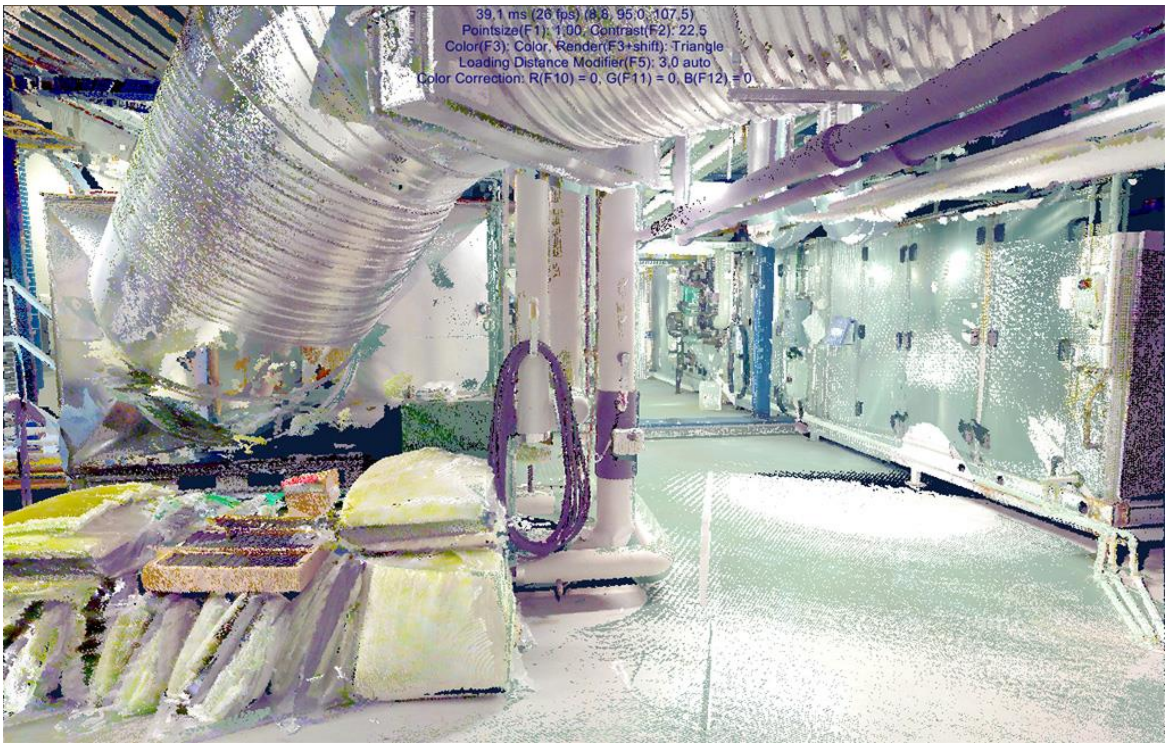
4.4 Pisteiden renderointi

Ensimmäisten onnistuneiden pisteiden renderointikokeiden myötä VR-laitteita käyttäen oli selvää, ettei pistepilven yksittäisten pisteiden piirto pistemäisinä ole kovin käyttäjäystävällinen vaihtoehto. Näkymän hahmottaminen oli vaikeaa ja viitteitä nopeasti syntyvästä pahoinvoinnista havaittiin.

Tämän ongelman ratkaisuyrityksenä päätettiin kokeilla pisteiden näkyvän koon suurentamista. Shader-koodin suorituskyky huomioon ottaen järkevin menetelmä tähän on piirtää jokaisen pisteen koh-

dalle sellainen kaksiulotteinen geometrinen kuvio, jonka sivut yhdistävinä pisteinä toimivien verteksien sijainnit on laskettavissa mahdollisimman yksinkertaisin laskutoimituksin. Ratkaisutapa on yleinen erilaisissa pistepilvien visualisointimenetelmissä, joissa halutaan välttää harvan pistepilven käytettävyysongelmat. Esimerkiksi Simon Fraiss käy selvityksessään (2017, 15-27) varsin syvällisesti läpi näitä menetelmiä erilaisilla pisteiden piirtotavoilla.

Käytännöllisiä kuvioita ovat hyvin suuren pistemäärän yhtäaikaisen piirtämisen yhteydessä kolmio ja neliö. Molempia kokeiltiin ja kumpikin vaihtoehto havaittiin tietyissä tilanteissa toista toimivammaksi vaihtoehdoksi. Tämän vuoksi molemmat vaihtoehdot ovat olleet pitkän aikaa mukana projektin kehitysversioissa, ja niiden välillä vaihtamiseksi kesken käytön ohjelmoitiin suora pikanäppäin. Alla olevassa kuvassa (KUVA 4) pisteistä on piirretty kolmioita joiden koko on fyysisen maailman mittakaavassa parin senttimetrin suuruusluokkaa.



KUVA 4. LIDAR-skannattua pistepilveä opinnäytetyössä kehitetyn ohjelmiston esittämänä

Tästä aiheesta on jäljemmässä osassa tarkempaa kuvausta. Aihetta avataan perusmuotoisella esimerkillä shader-koodin toiminnasta ja muistakin shader-koodin puolella tapahtuvista operaatioista kerrotaan jonkin verran yksityiskohtaisemmin.

4.5 Culling

Culling tarkoittaa tietokonepelien grafiikan yhteydessä kaikkia niitä eri menetelmiä, joilla vältetään laskemasta pelimaailman grafiikasta niitä osia mitkä eivät kuitenkaan näkyisi käyttäjälle. Koska pistepilven piirto tapahtuu tässä projektissa vain näytönohjaimella suoritettavassa shader-koodissa, ei itse pelimoottori ole tietoinen pistepilven osuudesta eivätkä sen tarjoamat valmiit culling-menetelmät ole siksi käytettävissä. Helppotajuisena esimerkkinä toimii ns. occlusion culling, joka on menetelmä

sellaisten pelimaailman osien jättämiseksi piirtämättä jotka sattuvat aina tietyllä hetkellä olemaan muiden pelimaailman osien peittämänä (Unity 2018a). Tämä tekniikka edellyttää pelimoottorin tietävän pelimaailman objektien väliset suhteet jotta peittotilanteet voitaisiin laskea ennakkoon.

Culling-menetelmien puuttuminen tarkoittaisi käytännössä, että aivan kaikki latausetäisyyden sisällä oleva pistepilvidata renderoitaisiin jatkuvasti, riippumatta siitä jääkö se pelimaailmaa näyttävän virtuaalisen kameran kuvakulman ulkopuolelle tai lähempien näkymän osien peittämäksi. Tämä tarkoittaisi näytönohjaimen grafiikkasuorittimen kuormituksen moninkertaistumista verrattuna tilanteeseen, missä vain näkökentässä oleva osuus näytönohjaimen muistiin ladatusta pistepilvidatasta piirrettäisiin. Tarve omien culling-menetelmien kehittämiselle otettiin tämän vuoksi huomioon jo varhaisessa vaiheessa kehitystyötä.

Kohdassa 4.2.2 kuvattu mittausasemakohtainen pistepilvidatan lataus korvaa tietyssä mielessä menetelmät joissa lähempien osien peittoon jäävä kauempi osuus jätettäisiin piirtämättä. Koska pistepilvidataa käsitellään mittausasemakohtaisesti, tulee aina yksittäisen osapistepilven sisältämien pisteiden raja vastaan siinä kohtaa missä mittauspisteestä katsoen laserskannaus kohtaa esimerkiksi seinän. Pelimoottorin kautta sijoiteltujen kiinteiden objektien luoma etäisemmän kohteen peittävyys korvaantuu laserskannerin toimintaperiaatteesta seuraten alkuperäisessä ympäristössä ilmenevällä fyysisten objektien toisiaan peittävyydellä.

Tasamittaisiksi kuutioiksi kaikki pistepilvimateriaali pilkkomalla eivät puolestaan yksittäisten latauslohkojen rajat enää vastaa yhden laserkeilauspisteen näkyvyyden rajoja. Vaihtoehtona tarjolle kuitenkin tulee mahdollisuus arvioida yksittäisitä latauslohkoista kuinka tiheään ne sisältävät pisteitä, eli kuinka tehokkaasti ne peittävät taakseen jäävät pistelohkot. Tällöin voidaan suhteellisen helposti luoda pistepilven lohkojen lataukseen menetelmä mikä ottaa tämän huomioon määritellessään lohkojen latauksen priorisointia.

5 PERUSTOIMINNALLISUUS

5.1 Pistepilvidatan ajonaikainen lataus

Seuraavaksi käsitellään pistepilven lohkojen latausta yleisesti, keskittyen tämän toteutustapaan teknisellä tasolla osana ohjelman ajonaikaista suoritusketjua. Tämä koskee samalla tavalla kaikkia aiemmin luvun 4.2 alla käsitellyistä vaihtoehtoisista menetelmistä latauslohkojen muodostamiseen.

5.1.1 Tiedostoformaatti ajonaikaisessa latauksessa

Tiedostojen lukemisessa ohjelman ajon aikana, sekä eräissä muissa niille tehtävissä operaatioissa käytetään PCL-kirjastoa C++ -kielellä kirjoitetun rajapinta-dll:n kautta. Tämä dll-tiedosto näkyy Unitylle pluginina (”Unity native plugin”) ja sen funktiot paljastetaan Unityn C# -koodille DllImport-määrityksillä.

PCL:lla on aiemmin kohdassa 3.2 mainittu oma pcd-päätteinen tiedostoformaattinsa, joka koostuu ascii-muotoisesta header-osiosta ja tätä seuraavasta binääridatasta. Binääridatan osuus on käytännössä suora muistilohkokopio keskusmuistiin siirrettävästä datasta, joten sen lukunopeus kiintolevyltä on suurin mahdollinen. (Point Cloud Library 2018b.)

5.1.2 Datan tuonti perusmuodossaan

Käytännössä tarve nopean lataamisen mahdollistavalle binääriformaatille tarkoittaa, että sovellukseen ladattava pistepilvidata työtiedostoina on muunnettava yllämainittuun PCL-kirjaston pcd-binääriformaattiin aina kun uutta materiaalia halutaan ladata sovelluksen käyttöön. Nuo työtiedostot ovat mahdollisimman lähellä LIDAR-laitteiden toiminnallista tasoa olevia ja sellaisina epäkäytännöllisen hitaita ladattavia.

Itse tiedostojen muuntaminen tapahtuu C++ -kielellä kirjoitetussa plugin-dll:ssa seuraavan periaatteen mukaan: Alkuperäisistä tiedostoista luetaan ensin header-osioista tiedostossa kuvattujen pisteiden lukumäärä ja yhteiseen rekisteröityyn koordinaatistoon muuntamisessa tarvittava muunnosmatriisi myöhempää käyttöä varten. Luodaan halutun tyyppinen PCL-pistepilviobjekti, luetaan alkuperäista ascii-dataa rivi kerrallaan ja asetetaan näin saadut tiedot pistepilviobjektin pisteiden arvoiksi. Kun koko alkuperäinen tiedosto on luettu ja sen tiedot on kirjoitettu pistepilviobjektiin, suljetaan tiedosto, luodaan tyhjä pistepilviobjekti ja kirjoitetaan alkuperäinen data muunnettuna ym. muunnosmatriisin mukaisesti haluttuun kohdekoordinaatistoon.

```
pcl::PointCloud<pcl::PointXYZRGBA> ptxcloud;
Eigen::Affine3f transform = Eigen::Affine3f::Identity();
// Tässä välissä kirjoitetaan muunnosmatriisidata transform-rakenteeseen
// ja täytetään pistepilviobjekti ptxcloud pisteiden datalla
pcl::PointCloud<pcl::PointXYZRGBA>::Ptr transformed_cloud(new pcl::PointCloud<pcl::PointXYZ-
RGBA>());
```

*pcl::transformPointCloud(ptxcloud, *transformed_cloud, transform);*

Yllä tiivistetty ote pistepilviobjektien ja muunnosmatriisin luomisesta, ja muunnosmatriisin soveltamisesta pistepilviobjektia vasten käyttäen PCL:n ja sen vaatiman Eigen-kirjaston valmiita työkaluja.

Kun lähdetiedostojen sisältämä data on muunnettu samaan koordinaatistoon ja tallennettu sovelluksen sisäisessä formaatissa, on vuorossa datan analysointi ja prosessointi kokonaisuutena halutun periaatteen mukaisesti. Mittausasemakohtaiset osapistepilvet joko jätetään sellaisinaan latausyksiköiksi, jolloin ne vain analysoidaan ja tämän tulokset tallennetaan, tai toteutetaan koko pistepilvikokonaisuuden kuutiointi kohdan 4.2.3 mukaisesti.

Kummankin vaihtoehdon mukaan edetessä luodaan tallennettessa tiedostoa sisäiseen latausformaattiin omat versionsa useammalle LOD-tasolle samoista osapistepilvistä. LOD-tasojen luomisessa hyödynnetään PCL:n voxelgrid-luokkaa. Tällä luodaan koko pistepilven kattava kolmiulotteinen kuutioista muodostuva verkko, jossa yhden kuution koko on sitä suurempi (ja näin ollen kuutioiden määrä pienempi) mitä epätarkempi LOD-taso on kyseessä. Jokaisen kuution sisältä lasketaan keskiarvo kaikista sen sisälle osuvista pisteistä sekä väriarvon että sijainnin suhteen ja korvataan ne yhdellä pisteellä joka saa tämän keskiarvon. Voxelgridin tiheys siis kuvaa korkeinta mahdollista lopputuloksena olevan pistepilven tiheyttä; On huomattava, että osa voxel-kuutioista jää kokonaan ilman pisteitä ja toisaalta lähdemateriaalin tiheimmillä alueilla keskiarvopiste muodostetaan suuresta määrästä pisteitä.

Datan tuonnin kokonaisuuteen kuuluva yksittäisten osapistepilvien tietojen analysoinnin tulos kirjoitetaan talteen tuontikansiossa sijaitsevaan json-tiedostoon. Kun tuonti eli mahdollinen kuutiointi, datan tallennus nopeasti ladattavaan muotoon ja yhteiseen koordinaatistoon, analysointi sekä LOD-tasojen luonti on tehty, voi materiaalin kopioida tuotuna kokonaisuutena kansiorakenteen säilyttäen toiselle koneelle ilman että tuontia tarvitsee tehdä uudelleen. Useamman LOD-tason vaatimista rinnakkaisista samaa aluetta kuvaavista datatiedostoista huolimatta tuonninjälkeinen kansiorakenne vie selvästi vähemmän tilaa kuin alkuperäinen tuotu data ptx-formaatissa. Epätarkemmat LOD-tasot nimittäin pudottavat pisteiden määrää hyvin radikaalisti, mihin niiden käytön hyöty etäämmällä havainnoijasta tietysti perustuukin, ja binäärimuoto itsessään mahdollistaa datan tallentamisen selvästi alkuperäistä ascii-pohjaista formaattia vähemmän tilaa vievästi.

5.2 Shader: Renderointi ja frustum culling

Tässä raportissa on aiemmassa kohdassa kuvattu, kuinka pistepilvidata välitetään näytönohjaimen muistissa sijaitseville buffereille. Shader-koodissa kyseisten bufferien sisältö viimein piirretään näytölle, mutta niille on tehtävä tiettyjä keskeisiä operaatioita.

Kokonaisuudessaan keskeisin näistä operaatioista on virtuaalimaailman globaalin koordinaatiston ja siellä sijaitsevan kameran (VR-lasien näyttämä suunta) välisen koordinaatiston sovittaminen toi-

siinsa. Ilman tätä vaihetta pistepilvidatan esittämä näkymä seuraisi aina mukana käyttäjän kääntäessä päätään. Unity tarjoaa shader-koodin kirjoittajille onneksi suoraan muunnosmatriisit maailman ja kameraprojektion välisiin koordinaatistomuutoksiin (Unity 2018b). Näiden soveltaminen koodissa kulloinkin renderointivuorossa olevalle pisteelle on helppoa ja suoraviivaista, joten kun asian merkitys on selvillä ei tästä asiasta seuraa sen suurempaa ongelmaa.

Samassa vaiheessa vertex shader -koodilohkossa on tarpeen myös laskea apumuuttujat geometry shader -lohkossa pisteistä piirrettävien primitiivien skaalaukseen. Niiden etäisyys kamerasta tarvitaan jotta ne voidaan piirtää oikean kokoisina eli kauemmaksi jäävät perspektiivin mukaisesti pienempinä. Samalla saadaan laskettua tarvittavat tiedot kameran kuvakulman ulkopuolelle jäävien pisteiden pois pudottamiseksi renderointiketjusta ennen kuin siirrytään suhteellisen paljon suorituskykyä syövään primitiivien luontiin. Nämä saadut arvot tallennetaan niitä varten luotavaan struktuuriin tms. nippuun muuttujia käytettäväksi seuraavassa koodilohkossa.

Esimerkin vuoksi lainaan Simon Fraissin kandidaattityössään (Fraiss 2017, 16-19) esittelemää menetelmää neliöiden piirtämiseen pisteistä geometry shader -lohkossa. Opinnäytetyöprojektissa sovellettiin alla kuvattua, suoraan Fraissin työstä lainattua menetelmää tietyin muutoksin. Kyseisestä julkaisusta löytyy allamainituilla sivuilla enemmänkin vaihtoehtoisia toteutuksia ja huomionarvoista pohdintaa.

```
[maxvertexcount(4)]
void geom(point VertexMiddle input[1], inout TriangleStream<VertexOutput> outputStream) {
        float xsize = _PointSize / _ScreenWidth;
        float ysize = _PointSize / _ScreenHeight;
        VertexOutput out1;
        out1.position = input[0].position;
        out1.color = input[0].color;
        out1.uv = float2(-1.0f, 1.0f);
        out1.position.x -= out1.position.w * xsize;
        out1.position.y += out1.position.w * ysize;
        //out2, out3 and out4 are calculated similarly, but with other +- combinations
        outputStream.Append(out1);
        outputStream.Append(out2);
        outputStream.Append(out4);
        outputStream.Append(out3);
}
```

Yksinkertaistetusti tässä lohkossa siis luodaan pisteen sijainnin ja halutun primitiivin koon perusteella neljä kulmapistettä, joiden mukaan neliö muodostuu. Vastaavalla tavalla voidaan luoda muitakin geometrisiä primitiivejä, esimerkiksi opinnäytetyön toteutuksessa on vaihtoehtoinen kolmioiden piirto pisteistä ja Unity-koodista käsin muutettavalla asetus-bufferiin kirjoitettavan struktuurin yhden boolean-arvon mukaan edetään sitä haaraa koodissa mikä tuottaa valinnan mukaisen piirtotuloksen.

5.3 Shader: Kuvan reaaliaikainen muokkaus

Kohdassa 5.2 mainitaan pisteiden piirto asetusbufferiin Unity-koodista käsin kirjoitettavan struktuurin yhden muuttujan arvon mukaisesti joko kolmioiksi tai neliöiksi. Samassa struktuurissa on myös muuttujat halutuille väri- ja kontrastikorjailuille. Nämä arvot luetaan ja niitä käytetään pistepilvidatan mukana oletuksena tulleiden RGB-väriarvojen muuttamiseksi halutuiksi. Tuloksena saatu väri asetetaan fragment shader -koodilohkossa juuri ennen näytölle piirtämistä samalla tavoin kuin jos kyseessä olisi alkuperäinen datan mukana tullut väriarvo.

Näkymän kontrastia lisätään kaavalla, mikä siirtää kutakin väriarvoa kauemmaksi sallittujen arvojen 0-255 puolivälistä ja suhteessa sitä enemmän mitä lähempänä puoliväliä ollaan. Oletusarvoksi laitettiin pieni kontrastin lisäys. Myös värikohtaisia arvoja voidaan koodissa muuttaa helposti Unity-koodista käsin shaderille välitettävien korjailukertoimien mukaan, ajatuksena että tilannetta voi näin parantaa, mikäli jokin tietty väri dominoi näkymää esimerkiksi taltioidussa fyysisessä tilassa vallinneen valaistuksen värin vuoksi.

Näkymän voi myös muuttaa kokonaan mustavalkoiseksi, mikä tapahtuu helposti laskemalla kaikkien värikomponenttien keskiarvo ja asettamalla tämä sama arvo kullekin värikomponentille. Tällöin kuvan muodostavien pisteiden kirkkaudet säilyvät samana kuin ennen mustavalkoiseksi muuntamista, mutta väri-informaatio katoaa keskiarvoistamisen myötä.

Koska lopulliset, korjaillut väriarvot lasketaan shader-koodissa jokaiselle pisteelle ja jokaisessa framessa erikseen, tapahtuvat muutokset reaaliaikaisesti asetusta muutettaessa. Nämä laskutoimitukset ovat niin kevyitä että mitään havaittavaa suorituskykyeroa ei saatu aikaan vaikka koko väri- ja kontrastikorjailujen koodi kommentoitiin pois vertailtavaksi muuten identtisellä shader-koodilla ja samalla käyttötilanteella. Testaaminen toteutettiin mittaamalla identtisissä tilanteissa ruudunpäivitysnopeus sekä korjailut sisältävällä koodiversiolla, että versiolla jossa tätä toiminnallisuutta ei ollut.

5.4 Shader: Raycast

Kehitettävänä olevan sovelluksen muissa kuin opinnäytetyön pistepilvitukeen liittyvissä ominaisuuksissa käytetään polygonimesheille mittaustyökaluja, jotka perustuvat Unity-pelimoottorin API:n monikäyttöiseen Physics.Raycast -metodiin. Unityn fysiikkamoottori ei kuitenkaan näe pistepilvien osuutta koska ne kulkevat Unityn läpi vain koordinaattidatana ja ne luodaan pelimaailmaan näytönohjaimella sinänsä pelimoottorista riippumattomina. Näin ollen on tarpeellista kirjoittaa tarvittava minimimäärä fysiikkatoiminnallisuutta suoraan samaan shader-koodiin mikä myöskin renderoi pistepilvet.

Raycast tarkoittaa käytännössä tilannetta, missä määritellystä lähtöpisteestä lähtee vektori määriteltyyn suuntaan (Unity 2018c). Kun vektori osuu johonkin pelimaailman objektiin, saadaan tuosta osumapisteestä tiedot. Pistepilville toteutetun korvaavan toiminnallisuuden tarkat yksityiskohdat jäävät

liikesalaisuuden piiriin mutta niiden toiminnan periaatetta voidaan hieman hahmotella karkealla tasolla.

Raycast-vektorin lähtöpisteen ja suunnan välittäminen Unityn C#-koodista shader-koodille näytönohjaimen muistibufferin kautta on suoraviivaista, ja se tapahtuu samalla tavalla kuin itse pistepilvidatan ja värikorjailuasetustenkin välittäminen. Shader-koodissa käsitellään jokaista pistepilven pistettä erikseen ja loogisesti pisteiden käsittely etenee kamerasta kauimmaisesta pisteestä kohti lähintä. Suoritusvuorolla aina kullekin pisteelle tehdään tarkistus toteuttavatko ne ehdon, mikä tässä tapauksessa on niiden sijainti mainitun raycast-vektorin reitillä. Jos kuitenkin raycast-vektoriksi on asetettu nollavektori koko tarkistus skipataan turhan laskennan välttämiseksi; Nollavektorilla ei ole suuntaa eikä pituutta, joten piste ei missään tapauksessa sijaitse sen kulkureitillä. Unity-koodista käsin mittausvektori myös kannattaa asettaa nollavektoriksi aina kun mittaus ei ole käynnissä, jotta turhalta tarkistuksien laskennalta vältyttäisiin.

Ongelmaksi tulee tähän asti kuvatulla toteutuksella se, että pisteellä mihin suhteessa tarkistus tehdään ei ole sijainnin lisäksi muita ulottuvuuksia. Kun vektorillakaan ei ole leveyttä, ei mittausvektori osuisi loogisesti ikinä yhteenkään pisteeseen, ja käytännössäkin se osuisi johonkin pisteeseen vain sattumalta silloin kun käytössä oleva liukuluvun tarkkuus sattuu loppumaan kesken hudin toteamiseksi. Tarkistuksessahan käytetään alkuperäisen pistedatan mukaista pisteen sijaintia eikä niitä pisteitä, mitkä piirretään näytölle geometry shader -lohkon mukaisen primitiivin luonnin myötä. Tämän vuoksi vektorista täytyykin luoda tarkistusten kautta käsitteellisessä mielessä halutun paksuinen palkki, ja paksuus sovittaa pistetiheyteen siten että se ei mahdu pisteiden välistä.

Jokainen piste mikä täyttää ehdon, eli on käsitteellisen tason palkin kulkureitillä, otetaan huomioon, ja se piste jonka etäisyys on pienin mittausvektorin lähtöpisteestä laskettuna, on raycast-tulos. Tämän pisteen koordinaatti ja välimatka raycast-lähtöpisteeseen asetetaan muistibufferiin, josta se luetaan Unityn koodissa ennen seuraavan framen piirtokäskyn antamista.

6 JATKOKEHITYS

Kirjoitushetkellä projektin kehitys jatkuu edelleen työsuhteessa kohti julkaisukelpoista versiota, jolloin se yhdistetään 3D Talo Finland Oy:n Design Space -suunnittelusovellukseen (3D Talo Finland Oy 2018). Tällä hetkellä erityisen huomion kohteena ovat C++ -kielellä kirjoitettavan datan tuontitoiminnallisuuden eri tiedostoformaattien tuen laajentaminen ja jo olemassa olevien optimointi tehokkaammiksi.

Koska vasta vähän aikaa sitten siirryttiin mittausasemakohtaisista latausyksiköistä yhdenmukaisiin kuutioihin tehokkaan tämän vaatiman muunnosalgoritmin kirjoittamisen myötä, on Unity-puolen latustoiminnallisuuden kirjoittaminen uusiksi edessä. Tähän on suunnitelmat valmiiksi hieman saman suuntaisina kuin aiemmin pisteiden renderoinnin yhteydessä mainitun Simon Fraissin kandidaattityössä (Fraiss 2017, 10-15.) mutta erona on suunnitelma käyttää pian julkaistavan Unity 2018.1:n kokonaan uutena ominaisuutena tuomaa ”Job System” nimen saanutta pelimoottorin sisäistä säikeistystoiminnallisuutta. Tämä oletettavasti tulee tarjoamaan huomattavasti selkeämmän ylläpidettävyyden kuin Unityn säikeistystuen puuttuessa kiertotienä käytettävä .NET säikeistystuki millä ei ole suoraa pääsyä Unity API:n ominaisuuksiin.

Kokonaisuutena jatkokehitykselle ei ole nähtävissä varsinaista päätepistettä, koska julkaisukelpoisen version jälkeen seuraa aktiivinen ylläpitokehitys sekä uusien ominaisuuksien kehittämistä. Kyseessä on sen verran uusi teknologia ja sen soveltamisala, että yritystoiminnassa ei voida jäädä lepäilemään itseensä tyytyväisenä ensimmäisen julkaisuversion jälkeen, mikäli halutaan pysyä mukana kilpailussa tai jopa sen kärjessä myös lähivuosina.

7 YHTEENVETO

Opinnäytetyön tuloksena on tuotekehityksen kannalta hyvässä vaiheessa oleva kehitysversio toimeksiantajan Design Space -sovellukseen liitettävästä pistepilvien suorarenderointituesta julkaistavaksi myöhemmässä vaiheessa. Opinnäytetyön toteutus täytti alkuperäisen toimeksiantonsa jo suhteellisen varhaisessa vaiheessa tämän raportin kirjoitushetkeen mennessä tehtyyn kokonaistyömäärään nähden, koska loppupäätelmä ensimmäisen vaiheen jälkeen oli tuotekehityksen jatkaminen palkaten opinnäytetyön tekijä julkaisuun tähtäävään tuotekehitykseen työsuhteessa.

Pistepilvituen julkaistavalta tuotteelta vaadittavan tasoisen toimivuuden toteutuskelpoisuudesta ei toimeksiantajalla ollut opinnäytetyöprojektia aloitettaessa varmaa näkemystä. Tutkittaessa aihetta löydettiin toimivat ohjelmistoarkkitehtuurilliset menetelmät, joiden tarkempi toteutus on hioutunut vähitellen monipuolisen kokeilun myötä. Alkuperäisiä tavoitteita on korjattu ja tarkennettu sitä mukaa kun sekä teknisten rajoitteiden että resurssikysymysten suhteen on saatu selville yksityiskohtaisempaa tietoa.

Toteutuksesta syntyi väistämättä modulaarinen kokonaisuus, joten yksittäisten osien toteuttamisessa on ollut helppoa testata erilaisia ratkaisumalleja, valita näistä parhaiten toimivat, ja hioa niitä rinnakkain testattavilla edelleen aiemmin valituista parhaista malleista periytyvillä erilaisilla ratkaisuvaihtoehdoilla. Tässä suhteessa kehityskaari muistuttaa vahvasti ketterän ohjelmistokehityksen menetelmiä. Erona on vain se, että noiden menetelmien teoriassa edellytetään samaan osakokonaisuuteen erikoistunutta kehitystiimiä yhden kehittäjän sijaan. Muilta piirteiltään kehitysprojektin etenemistä voisi kuvata nykyisin trendikkäillä käsitteillä kuten ”Scrum” tai ”Kanban” joista jälkimmäisen mukainen Kanban-taulu onkin ollut sovellettuna versiona käytössä projektin seurannassa.

Opinnäytetyön merkitystä korostaa, ettei samalle kohderyhmälle tarkoitetussa vastaavan mittakaavan ohjelmistotuotteessa kuin Design Space kokonaisuudessaan tulee olemaan, ole ennestään markkinoilla kovinkaan toimivaa pistepilvien suorarenderoinnin tukea. Kysyntää tällaiselle toiminnallisuudelle kuitenkin olisi runsaasti. Toimeksiantajan asiakkaina olevat mm. rakennus- ja koneteollisuuden yritykset ovat osoittaneet selvän kiinnostuksensa näille ominaisuuksille jo varhaisten kehitysversioiden esittelymateriaalin pohjalta. Mahdolliset sovellusalat ovat kuitenkin paljon laajemmat, sisältäen teollisuuden eri osa-alueet raskaaseen teollisuuteen asti mutta ulottuen myös esimerkiksi maisema- ja kaupunkisuunnitteluun, geologiseen tutkimukseen sekä maanrakennukseen.

Oppimisen kannalta opinnäytetyön aiheeseen perehtyminen ja projektin edistymisen aikaan kohdatut ohjelmoinnilliset haasteet ovat olleet todella hyödyllisiä. Ennestään tuntemattomien teknologioiden omaksuminen on ollut välttämätöntä ja työskentely monipuolista. C++ -ohjelmointissa algoritmien suunnittelu ja eri matemaattisten kirjastojen hyödyntäminen, sekä shader-ohjelmoinnissa vuorovaikutteisen toiminnallisuuden toteuttaminen ovat matemaattisesti varsin raskaita osa-alueita. Virtuaalitodellisuuslaitteiston käyttäjien käyttäjäkokemuksen optimointi taas on täysin erilainen puoli projektista, ja lopulta kaikki sinänsä hyvin erilaiset osa-alueet nivoutuvat yhdeksi kokonaisuudeksi

missä jossain osa-alueessa tehdyt valinnat vaikuttavat kaikkialla. Itsessään työskentely innovatiivisessa, uuden teknologian aallonharjalla toimivassa startup-yrityksessä on edistänyt oman työmotivaationi hallintaa ja taitoa ajatella kuinka uusien teknologioiden hyödyntäminen on tuotteistettavissa mahdollisia asiakkaita hyödyttäviksi ohjelmistoiksi.

LÄHTEET JA TUOTETUT AINEISTOT

FATAHALIAN, Kayvon 2011. How a GPU Works. Saatavissa: https://www.cs.cmu.edu/afs/cs/academic/class/15462-f11/www/lec_slides/lec19.pdf

FRAISS, Simon Maximilian 2017. Rendering Large Point Clouds in Unity. Saatavissa: https://www.cg.tuwien.ac.at/research/publications/2017/FRAISS-2017-PCU/FRAISS-2017-PCU-thesis.pdf

LEVOY, Marc ja WHITTED, Turner 1985. The Use of Points as a Display Primitive. Saatavissa:

http://graphics.stanford.edu/papers/points/point-with-scanned-figs.pdf

MAANMITTAUSLAITOS 2018. Laserkeilausaineisto [Verkkoaineisto]. [Viitattu 2018-05-03]. Saatavissa: https://www.maanmittauslaitos.fi/kartat-ja-paikkatieto/asiantuntevalle-kayttajalle/tuotekuvaukset/laserkeilausaineisto

NALE, Alberto 2014/2015. Design and development of a generalized LIDAR point cloud streaming framework over the web. Saatavissa: http://tesi.cab.unipd.it/47156/1/Nale_Alberto_1035304.pdf

PCL – POINT CLOUD LIBRARY 2018a. About: What is PCL? [verkkoaineisto]. [Viitattu 2018-04-12].

Saatavissa: http://pointclouds.org/about/

PCL – POINT CLOUD LIBRARY 2018b. Documentation: The PCD (Point Cloud Data) file format

[Verkkoaineisto]. [Viitattu 2018-05-08]. Saatavissa: http://pointclouds.org/documentation/tutorials/pcd_file_format.php

UNITY 2018a. Documentation: Occlusion Culling [Verkkoaineisto]. [Viitattu 2018-05-08]. Saatavissa: https://docs.unity3d.com/Manual/OcclusionCulling.html

UNITY 2018b. Documentation: Built-in shader variables [Verkkoaineisto]. [Viitattu 2018-05-08]. Saatavissa: https://docs.unity3d.com/Manual/SL-UnityShaderVariables.html

UNITY 2018c. Documentation: Physics.Raycast [Verkkoaineisto]. [Viitattu 2018-05-08]. Saatavissa: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

WANG, Jun ja SHAN, Jie 2005. Space-Filling Curve Based Point Clouds Index. Saatavissa: http://www.geocomputation.org/2005/WangJ.pdf

3D TALO FINLAND OY 2018. Design Space – Design in Virtual Reality [verkkoaineisto]. [Viitattu 2018-04-12]. Saatavissa: http://3dtalo.fi/en/design-space/